package com.twitter.ann.common

import com.twitter.ann.common.EmbeddingType.EmbeddingVector

import com.twitter.storehaus.{ReadableStore, Store}

import com.twitter.util.Future

// Utility to transform raw index to typed index using Store

object IndexTransformer {

/\*\*

\* Transform a long type queryable index to Typed queryable index

\* @param index: Raw Queryable index

\* @param store: Readable store to provide mappings between Long and T

\* @tparam T: Type to transform to

\* @tparam P: Runtime params

\* @return Queryable index typed on T

\*/

def transformQueryable[T, P <: RuntimeParams, D <: Distance[D]](

index: Queryable[Long, P, D],

store: ReadableStore[Long, T]

): Queryable[T, P, D] = {

new Queryable[T, P, D] {

override def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

): Future[List[T]] = {

val neighbors = index.query(embedding, numOfNeighbors, runtimeParams)

neighbors

.flatMap(nn => {

val ids = nn.map(id => store.get(id).map(\_.get))

Future

.collect(ids)

.map(\_.toList)

})

}

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

): Future[List[NeighborWithDistance[T, D]]] = {

val neighbors = index.queryWithDistance(embedding, numOfNeighbors, runtimeParams)

neighbors

.flatMap(nn => {

val ids = nn.map(obj =>

store.get(obj.neighbor).map(id => NeighborWithDistance(id.get, obj.distance)))

Future

.collect(ids)

.map(\_.toList)

})

}

}

}

/\*\*

\* Transform a long type appendable index to Typed appendable index

\* @param index: Raw Appendable index

\* @param store: Writable store to store mappings between Long and T

\* @tparam T: Type to transform to

\* @return Appendable index typed on T

\*/

def transformAppendable[T, P <: RuntimeParams, D <: Distance[D]](

index: RawAppendable[P, D],

store: Store[Long, T]

): Appendable[T, P, D] = {

new Appendable[T, P, D]() {

override def append(entity: EntityEmbedding[T]): Future[Unit] = {

index

.append(entity.embedding)

.flatMap(id => store.put((id, Some(entity.id))))

}

override def toQueryable: Queryable[T, P, D] = {

transformQueryable(index.toQueryable, store)

}

}

}

/\*\*

\* Transform a long type appendable and queryable index to Typed appendable and queryable index

\* @param index: Raw Appendable and queryable index

\* @param store: Store to provide/store mappings between Long and T

\* @tparam T: Type to transform to

\* @tparam Index: Index

\* @return Appendable and queryable index typed on T

\*/

def transform1[

Index <: RawAppendable[P, D] with Queryable[Long, P, D],

T,

P <: RuntimeParams,

D <: Distance[D]

](

index: Index,

store: Store[Long, T]

): Queryable[T, P, D] with Appendable[T, P, D] = {

val queryable = transformQueryable(index, store)

val appendable = transformAppendable(index, store)

new Queryable[T, P, D] with Appendable[T, P, D] {

override def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

) = queryable.query(embedding, numOfNeighbors, runtimeParams)

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

) = queryable.queryWithDistance(embedding, numOfNeighbors, runtimeParams)

override def append(entity: EntityEmbedding[T]) = appendable.append(entity)

override def toQueryable: Queryable[T, P, D] = appendable.toQueryable

}

}

}